Master's Thesis

ScanHash: Scannable Dynamic Hashing

Songyi Lee

Department of Electrical and Computer Engineering
(Computer Science and Engineering)

Graduate School of UNIST

2020

ScanHash: Scannable Dynamic Hashing

Songyi Lee

Department of Electrical and Computer Engineering
(Computer Science and Engineering)

Graduate School of UNIST

ScanHash: Scannable Dynamic Hashing

A thesis/dissertation
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Songyi Lee

12/16/2019

Approved by

Advisor
Young-ri Choi

ScanHash: Scannable Dynamic Hashing

Songyi Lee

This certifies that the thesis/dissertation of Songyi Lee is approved.

12/16/2019 of submission

signature

Advisor: Young-ri Choi

signature

Sam H. Noh: Thesis Committee Member #1

signature

Beomseok Nam: Thesis Committee Member #2

three signatures total in case of masters

Abstract

Indexing structure is getting important as the storage device is getting faster. Persistent memory such as Intel Optane DC Persistent Memory which is now commercially available is one of emerging fast storage devices that is byte-addressible, non-volatile and it has comparable access latency to the DRAM. On the other hand, traditional relational operation database has limitations to store large-scale data in terms of performance and scalability because of its relation. To overcome the limitations, a non-relational database called NoSQL such as key-value stores came out to handle big-data which is nowadays widely adopted for the commercial database system like MongoDB and it is necessary to have efficient indexing to provide fast performance. In this work, we present a novel dynamic hashing technique, *ScanHash*, that gets benefit fast point query of hash table index with sorting buckets in the natural order of keys to support scan operations. ScanHash provides efficient point query using based on the extendible hash table structure and scan query by sorted bucket with natural key order. To avoid the problem caused by the skewness of key distribution, we suggest the key remapping function replacing the hash function with the approximated cumulative distribution function. In our experimental results, ScanHash shows overall better performance than the state-of-the-art B+-tree for persistent memory.

Contents

List of Figures

I Introduction

As the storage becomes faster and data size grows rapidly, the limitations of a relational database management system that is traditionally used to store and manage data become severe problems to handle large-scale data on the fast storage device such as persistent memory that is one of the fast storage devices which is byte-addressable and non-volatile with the access latency comparable to DRAM. The relation makes the entire database heavy and operations slow in the relational database. To overcome the limitation, the NoSQL database excluding the relation from the database that is lightweight, fast and simple database came out.

Key-value store is one of the NoSQL database that is actively researched and widely adopted to the various commercial platform including amazon web services that providing MongoDB which is a key-value store. It manages the key and value pair as an object with an indexing data structure to find the specific object like a hash table and B+-tree depending on its usage. Although the hash table outperforms the other data structure in point query operation, the database which needs to support scan operation has to use a scannable indexing structure because it does not support scan operation.

As the query operation become simple and fast, the efficiency of the indexing structure is getting important. Some researches suggest that the indexing data structures using both hash table and secondary scannable data structure such as B-tree or Skiplist [1] [2] to have the benefit of fast point query of the hash table while supporting scan operation. However, indexing data structure having two indexing data structures has to synchronize two different data structures to provide consistent result and the slower data structure become a bottleneck, so they hide its bottleneck by buffering the queries and flushing it into the scannable index in the background.

In this work, we propose the scannable hash table without a secondary scannable data structure, named ScanHash, based on the extendible hash table using the prefix of the key as a directory index bits and maintaining sorted bucket by a natural key order to support the scan operation. Point query works as an extendible hash table and range query works as a leaf node of B+-tree with directory instead of an internal node. We also suggest the key remapping function which is cumulative distribution function replacing the hash function that distributes key uniformly to avoid the collision while the natural order of key is preserving under the assumption of distribution. It is designed for persistent memory guaranteeing consistency with 8 bytes atomic writes under the system crash.

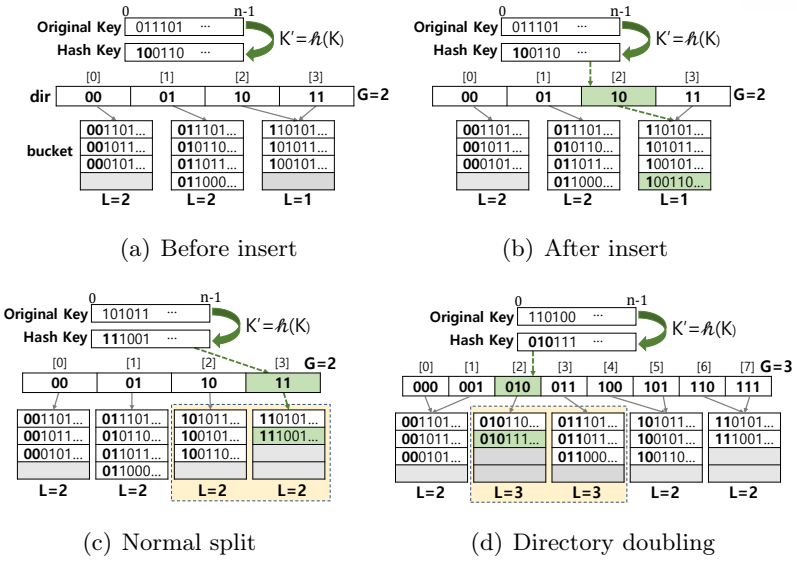
The main contributions of this work are as follows:

- We design a novel scannable dynamic hashing index that inherits the advantages of hashing for fast insert and search, while supporting efficient sequential processing of keys. We propose to use remapped keys, instead of hash keys, to preserve the natural order of keys.
- To maintain sorted buckets, we adapt Failure-Atomic ShifT algorithm [3], a technique that provides an efficient key shifting scheme and that tolerates transient inconsistent states

caused by system crash. We also integrate FAST with CCEH [4], a variant of extendible hashing optimized for persistent memory.

- To distribute keys uniformly, we develop a key mapping mechanism that periodically checks if the current key distribution is non-uniform, identifies the type of non-uniform distribution to generate a remapping function, and finally, construct a new directory based on a new remapping function.

- We implement ScanHash based on CCEH [4] and FAST [3]. We evaluate the performance of ScanHash by comparing with CCEH and the fast and fair B+-tree algorithm [3] using three different widely used key distributions, exponential, normal, and gamma distribution. Our experimental results demonstrate that ScanHash provides higher performance for insert, search, and scan operations, compared to the fast and fair B+-tree.



(a) Before insert (b) After insert

(c) Normal split (d) Directory doubling

Figure 1: Extendible hashing

II Background

2.1 Extendible Hashing

ScanHash is based on the extendible hashing [5] which is a dynamic hashing scheme that can significantly reduce rehashing overhead by dynamically grouping hash keys using some bits of the key. It uses prefix or suffix bits of the hash key as an index and keys with the same index are gathered in the same bucket. The extendible hash table consists of a Directory and several Buckets. The directory is an array of indexes which is prefix or suffix of key pointing to the corresponding bucket. The number of bits used as an index in the directory is called *global depth*. Each index pointer points a bucket while a bucket can be pointed by multiple index pointer. A bucket is a structure containing keys and value pointers as a pair with additional information. Keys in a bucket have common bits and the number of bits is called *local depth*. The local depth is always less or equal than the number of global depth.

Figure 1 shows that insert operation in the extendible hash table. In the state 1(a), when a new key is inserted, the raw key is converted to the hash key by a hash function. Prefix or suffix of the hash key is used as an index to find the corresponding bucket with common local depth bits. In the bucket, it scans the bucket to find the empty slot. If there is no place to put a new key, then it splits the bucket into two buckets and moves keys into the appropriate bucket. Note that two buckets created by the split operation have one additional local depth bit. When local depth is the same as global depth, it increases global depth, doubling the size of the directory to use an additional binary bit. 1(d) shows the directory doubling with the prefix index. The bucket containing key and value pairs remains as before except for the bucket that triggers split operation, doing efficient rehashing for the collision.

2.2 Persistent Memory

Byte-addressable and non-volatile new memory such as phase-change memory(PCM) [6] and STT-MRAM [7] is emerging including now commercially available Intel Optane DC Persistent Memory [8]. Persistent memory(PM) is projected to have slower but still comparable latency to DRAM and higher density than DRAM. It is generally accepted that the failure of atomic granularity for writes to PM is 8 bytes while a cacheline size is 64 bytes [9, 10]. Since memory writes to PM with a cacheline unit can be reordered, we need to guard the ordering of writes to PM using memory instructions and cache flush instructions such as CLFLUSH and MFENCE instruction [3, 9–12]. Moreover, for data whose size is larger than 8 bytes, we need to ensure the consistency of data using several techniques.

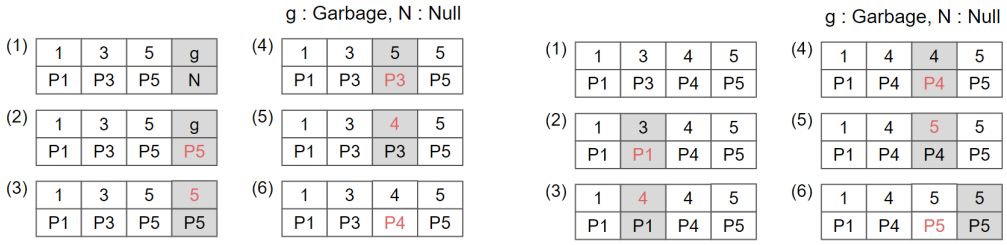
2.3 CCEH

Cacheline-Conscious Extendible Hashing(CCEH) [4] is one of the state-of-art indexing structure for persistent memory based on the extendible hashing optimizing cacheline access efficiently. The structure of CCEH is similar to the structure of the extendible hash table using prefix bits as the index except they use *Segment* as a primary bucket containing cacheline-sized secondary bucket. Since the MSBs are the index of the directory, they reduce the overhead of split and the directory can be recovered after the system crash by building a binary buddy tree which reflects the history of the split. When directory doubling occurs, adjacent indexes of the doubled directory have the same bucket pointer except for the bucket which triggers the split operation. For example, if the difference between global depth and local depth of a bucket is 1, then the bucket is pointed by two adjacent indexes. Therefore, it can recover the directory by scanning the directory in one direction and compare the difference between global depth and local depth of the bucket to the number of index pointer pointing the bucket. ScanHash also uses a similar approach of a binary buddy tree to build a new directory that is described in section 3.3.

2.4 FAST algorithm

Failure-Atomic ShifT(FAST) algorithm is devised for a B+-tree for persistent memory [3] where all pointers in B+-tree nodes are unique and maintained in sorted order. It minimizes the number of CLFLUSH and MFENCE operations that are expensive guaranteeing that a read transaction can detect inconsistent states.

Figure 2 shows each step of the insertion of a key 4 and corresponding value point P4 into a bucket and the deletion of key 3 with the FAST algorithm. The box with gray background in figure means an inconsistent state, in other words, the pointer is the same as a left pointer or Null. When a new key is inserted into a node, it first checks whether a free slot exists and node split occurs if it does not have a free slot. From the right side of the node in Figure 2(a), the value pointer of the key-value pointer pair is shifted from left to right first and the key is shifted from left to right. In the state 2 and 3 of Figure 2(a), the read transaction always read correct



(a) Inserting (4, P4) (b) Deleting (3, P3)

Figure 2: FAST algorithm

state. In the state 4, although the read transaction is searching for the key 5, the left key 5 is detected as an inconsistent state because the value pointer is the same as the left pointer P3. After shifting existing keys, new key 4 is inserted first and pointer P4. Deletion is the reverse of insertion shifting key first and value pointer after from right to left. ScanHash also uses the FAST algorithm to shift the key-value pointer pairs in the bucket efficiently.

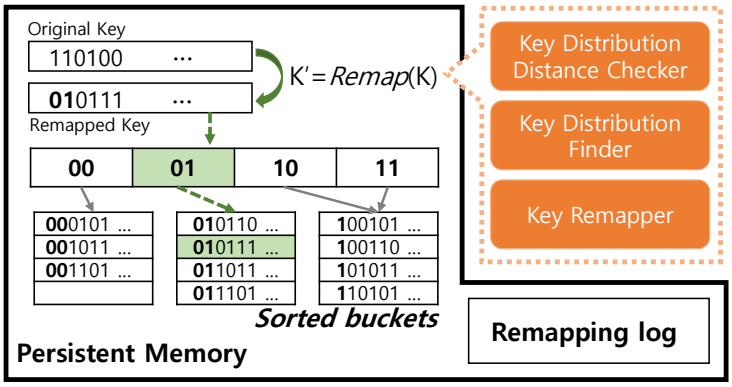


Figure 3: ScanHash architecture

III Overview

In this section, we describe the overview of ScanHash that is a scannable dynamic hashing based on the extendible hashing with sorted buckets in the natural order of raw keys using remapping function instead of hash function to reduce collision and rehashing overhead.

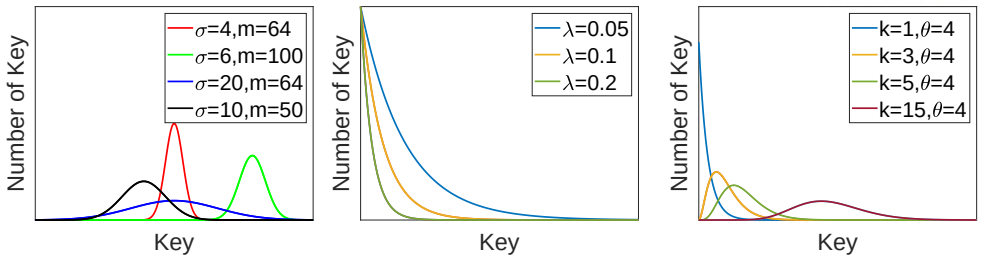
3.1 ScanHash Structure

ScanHash has similar to the extendible hashing except it uses another directory for the remapping process. Figure 3 shows the architecture of ScanHash. Note that it can be used for the volatile memory although current ScanHash is implemented for persistent memory. The directory is similar to the directory of extendible hash table except that ScanHash cannot use LSB of key as an index because of its sorted order. Each bucket has a fixed number of slot for key and value pointer pairs. It maintains 8 bytes key and 8 bytes corresponding value pointer pairs in sorted order of raw key.

The main idea of ScanHash is to use the key in the natural order with sorting to provide scan operation. In the hash table, it uses uniformly distributed hash key converted from key by a hash function to avoid the collision that triggers expensive rehashing. However, hash function does not preserve the natural order of raw keys. Instead of hash function, we use remapping function to uniformly distribute keys while preserving the natural order. We use *cumulative distribution function* as a remapping function instead of hash function where each value of point represents the sum of probability distribution function until the point and which is uniform distribution.

3.2 Key Distribution

Since the key is discrete and unique in the indexing data structure, intuitively, the number of keys smaller than specific key is monotonically increasing for each key like the cumulative distribution function of uniform distribution. Therefore, all the keys are mapped to the cumulative distribution function of the key which preserves the natural order of raw key.



(a) Normal (b) Exponential (c) Gamma

Figure 4: Key distributions

It can be proven that the cumulative distribution function F_X which is continuous and increasing function is uniform distribution [**?**]. Define $Z = F_X(X)$ and U is a uniform random variable that takes values in [0,1],

$$F_Z(x) = P(F_X(X) \leq x) = P(X \leq F_X^{-1}(x))$$
$$F_U(x) = \int_R f_U(u)du = \int_0^x du = x$$
$$\therefore F_Z(x) = F_U(x)$$

for every x $\in$ [0,1].

However, the cost to calculate exact cumulative distribution function from the set of given discrete keys are expensive because it requires scanning from the smallest key to the specific key for every query. On the other hands, some of continuous distribution such as normal and exponential distribution has formula of the cumulative distribution function using the parameter of distribution. To overcome the high overhead to calculate cumulative distribution function from discrete key, we first assume that key distribution is one of classified continuous distribution that the formula of cumulative distribution function is well known, in this work, normal, exponential, and gamma distributions as shown in the Figure 4 that are commonly used and observed in the natural and social science, and modified the continuous cumulative distribution function calculated by given formula of cumulative distribution function to discrete. By doing so, it is possible that two different discrete keys are mapped to the same remapped key since it ignores value below the decimal point. Even though it increases the chance of collision, it is acceptable because remapped key is only used to find directory index as hash function returning same hash value to the different key.

3.3 Remapping Process

If the distribution of key is skewed, in other words, most of the keys are in certain range of key space, most of the raw keys will have common MSBs makeing too frequent collision. For example, if we have bucket with 10 slot and we insert 11 keys from 0 to 10(1010(2)) when size of key is 64 bits, the global depth have to be 61 to insert key 10 because 60 bits of keys are

equal. Since the size of directory is same as 2^{global_depth}, we need to have 2^{61} indexes containing 8 bytes pointer which is 16 Exbibyte that is not available now.

To avoid this situation, ScanHash provides key remapping that consists of key distribution distance checker to check how far current key distribution is from a uniform distribution periodically or when too many consecutive directory doubling occurs, key distribution finder to find the appropriate distribution and its parameters of the current key distribution when distance checker decided that the current distribution is not uniform, and key remapper to build new directory with cumulative distribution function computed by distribution finder.

IV ScanHash

In this section, we describe basic operations and ScanHash with sorted bucket not considering the key remapping occurs or distribution changes. In addition, we describe the concurrency of ScanHash with lock-based insert and delete operations and lock-free search operation.

4.1 Operations with FAST algorithm

ScanHash provides basic operations Insert, Delete, Search, and Scan. Update works similar to Insert for the key that exists in the bucket, that it replaces the old value pointer to the new value pointer, but ignores the key that does not exist in the hash table. Therefore, we do not include a detailed explanation of the Update operation in this section.

Insert

When a new key is inserted, it converts the key into a remapped key using given remapping function, key itself assuming the distribution is uniform at first and finds the index of the directory using most or least significant bits. In the corresponding bucket that is pointed by the pointer located in the index of the directory, it first scans the bucket to find the inserted key to update in-place and persist its cacheline if the key already exists. When the key does not exist in the bucket and the number of key and value pairs in the bucket is less than the capacity of the bucket, it shifts keys and values from right to left until its appropriate position to insert new key preserving the natural order using FAST [3] algorithm. Otherwise, we split the bucket into two buckets with an additional one bit for local depth. If the one bit increased local depth is less than or equal to the global depth, then it updates the corresponding directory indexes. For the case that the local depth becomes greater than global depth after the split, we build the new directory with doubled directory size. After the directory indexes are set, we first persist the directory and indexes and atomically update the directory pointer and persist the directory pointer.

Delete

Delete operation works the same as insert operation before shifting the key and value pointer pairs in the bucket with the reverse of shifting for the insert operation. The key is converted to the remapped key using given remapping function, uniform at first, and the MSBs of global depth is used as an index of the directory. In the bucket, it first scans the bucket to find the key to delete before shifting for the case that the key does not exist to avoid unnecessary shifting. If it finds the key in the bucket, it overwrites from the position of the key the value using the value of the left key making the key as an invalid key and shifts key and value from right to left in a FAST way.

4.2 Concurrency and Consistency

Lock-based Operations

Operations that require bucket shift and bucket split must get a lock of the corresponding bucket before it shifts key and value pointer pairs. The directory has a lock for write when bucket split occurs to avoid the situation such that a thread updates the old directory while a new directory with the doubled size is copying the old directory. Note that it is possible to read the directory index while directory indexes are updating because search operation can check the prefix of the key is the same as what a key in the bucket should be and directory pointer is changed atomically for the directory doubling.

Lock-free Operations

The procedure to find the index of the directory is the same as insert and delete operations. In the bucket, it scans the bucket to find the given key and, if the key is found, compares the value to the previous value to check whether the key is invalid or not.

When two threads access the same bucket, the direction to read bucket should be the same as the direction to shift not to miss a shifted key. For example, the insertion thread shifts key and value from left to right as figure 2(a), so read thread has to read from left to right direction. Otherwise, it is possible to skip the shifted key. When a thread is deleting key 3 as in the figure 2(b), if a read thread finding key 4 searches the bucket from left to right and sleeps at the second stage, while deleting thread is working, then, although the key 4 exists in the bucket, because read thread restarts from the third slot, it will return the result that key 4 does not exist which is not true. To prevent misreading, each bucket has a direction indicator that is set when insert or delete thread access to the bucket. If the key is not found in the bucket, it compares the key has the same prefix with the bucket to check the split operation occurs while reading.

Scan operation is to scan from the given starting key or smallest key bigger than starting key until it reads N keys or it reaches the end. A search operation is equal to a scan operation scanning a single key. It first finds the index of directory and bucket using the remapped key as it does in the other operations and scans the bucket to find the starting key or smallest key bigger than the starting key. It compares the pointer to the previous pointer to detect the inconsistent state for each key and value pointer pairs. A bucket is sorted and has a sibling pointer that is updated when bucket splits not to read unnecessary directory indexes.

V Key Remapping

In this section, we describe three non-uniform distributions that we used in this work, normal, exponential, and gamma distributions, and how key remapping works with three different components, key distribution distance checker, key distribution finder, and key remapper. Although we use only three widely used distributions, another distribution can be added because the idea of ScanHash is to use the remapping function which is increasing the cumulative distribution function of key distribution instead of a hash function.

5.1 Key Distribution and cumulative distribution function

Three non-uniform distribution that are commonly used and observed in natural and social sciences and uniform distribution have their own characteristics that distinguishes it from the others and their parameter representing their distribution. For example, cumulative distribution function of normal distribution has fomula $\frac{1}{2}[1+erf(\frac{x-\mu}{\sigma\sqrt{2}})]$ where μ represents mean, σ is variance of the distribution, and erf means error function such that $erf(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$. If we have enough information to calculate the formula, we can get the cumulative distribution function value of key directly.

However, it is expensive to calculate functions like error function and exponential function in the cumulative distribution function formula to use as a remapping function that is used for every query. To remedy this problem, we convert the cumulative distribution function to a set of linear functions that approximate the cumulative distribution function based on the linear interpolation. We divide the entire key range into multiple sub-ranges that containing the same number of keys using the quantile function that is the inverse of the cumulative distribution function of the distribution. The range of cumulative distribution function is from 0 which represents key does not exist until this point to 1 that represents all keys exists before the point. For example, if we divide the key range into 10 sub-ranges, the difference between the results of the cumulative distribution function of the first key in the sub-range is 0.1. In a reverse way, we use the inverse of the cumulative distribution function, the quantile function, to find the first key in the sub-range. Since the purpose to use cumulative distribution function is to distribute keys uniformly to the entire keyspace, we stretch the range to fit the entire keyspace. Finally, we build linear functions connecting the first key in the sub-range and its cumulative distribution function value. It is acceptable to approximate the cumulative distribution function until the number of keys with a common prefix index is less than the capacity of the bucket.

5.2 Key Distribution Distance Checker

We assume that the key distribution is uniform at first, so we use an identical function as the remapping function. It is not always true the key distribution is uniform, so ScanHash has a key distribution distance checker to detect the non-uniform key distribution based on

the variance that is the squared standard deviation representing how many keys are far from the mean value. Since the uniform distribution with a key range from 0 to 2^{key_bits} has fixed variance, we compare the calculated variance of current key distribution and uniform to check how distribution is far from the uniform periodically. In addition, if the key distribution is too skewed, a lot of key shares common prefix global depth bits and they are inserted into the same bucket triggering too frequent directory doubling before a periodic checker is called. Considering the main purpose of remapping is to make the distribution uniform to reduce the split and huge directory overhead, we trigger the key distribution distance checker when the number of consecutive directory doubling is larger than some threshold.

5.3 Key Distribution Finder

If the key distribution distance checker detects that the current distribution is not uniform, then it calls a background thread to remap consisting of key distribution finder and key remapper. The key distribution finder finds that which distribution between exponential, normal, and gamma is the current distribution using mean and variance. After the distribution is decided, it builds a set of linear functions approximates the cumulative distribution function of the distribution.

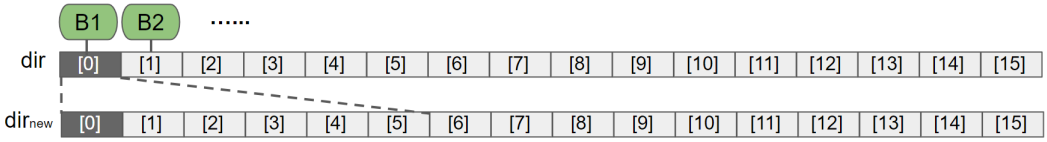
5.4 Key Remapper

Using given information calculated in the distribution finder, keys in the old directory are adjusted with the new remapping function. It is similar to the directory doubling with rehashing keys in the background. The key remapper first builds the new directory and then adjust keys.

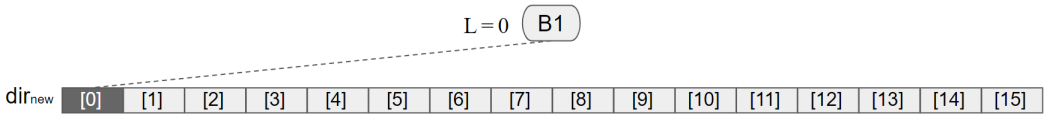
Directory builder

Since the remapping function is changed, directory indexes are also changed. Directory builder builds the same capacity as the old directory, so we block the directory doubling during the new directory builder is building the directory to prevent the change of directory capacity. If we allow the doubling of old directory during remapping, since the distribution is skewed triggering unnecessary directory doubling at the point the background remapping function is launched, it can occupy the memory with huge directory interrupting not only remapping but also entire system.

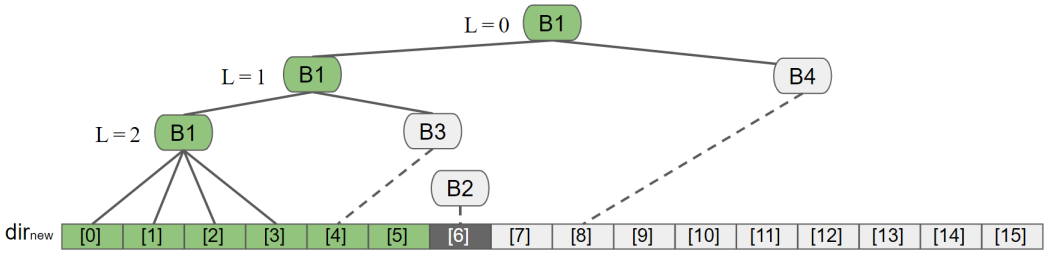
Figure 5 shows the process of building new directory. From the given remapping function, we calculate the remapped index from the smallest keys of each bucket. The index is the same as the global depth prefix of keys in the corresponding bucket, so we add 0s to the index bit to make its size equal to the key size for the minimum key of the bucket. As in the figure 5(a), dir[0] can be mapped multiple indexes of new directory. The problem is that a bucket always splits into two buckets in extendible hashing because it uses one additional binary bit for split, but it can happen that the smallest key in dir[0] is remapped to the $dir_{new}[0]$ and the smallest key in dir[1] is remapped to the $dir_{new}[6]$. From the split history, the directory must have a bucket at



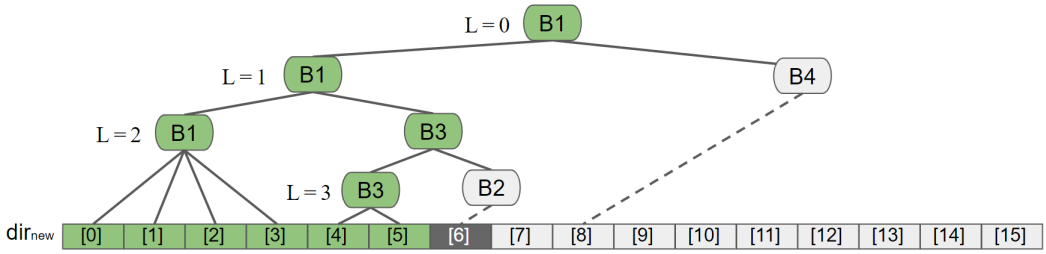
(a) Mapping of B1 from the old directory to the new directory



(b) Initial state of the new directory



(c) Computation of the stride for B1 at the new directory



(d) Computation of the stride for B3 at the new directory

Figure 5: New directory builder

$dir_{new}[8]$ which has bucket with keys of prefix 1 and $dir_{new}[4]$ with prefix 01 to have $dir_{new}[6]$. Considering the property of split history, building new directory for the new remapping function is similar to building split history from the bucket location. This procedure is similar to the recovery of CCEH using buddy tree except that its tree structure is already built and might have an error.

New directory starts from the figure 5(b) with a single bucket connected to first index with local depth 0 which represents the first state of directory without pre-allocation of bucket. The black box of figure means the index that new bucket for remapping will be created. For example, $dir_{new}[0]$ index is black in figure 5(b) because the smallest key of first index dir[0] is 0 which will be remapped to the $dir_{new}[0]$. After dir[0] is fixed, we move to the next bucket and calculate the remapped index of the smallest key of next bucket, dir[1] and $dir_{new}[6]$ in figure 5. From the state of figure 5(b), single directory doubling makes two buckets with local depth 1, named B1 and B4 in the figure 5(c). The bucket B1 covers from $dir_{new}[0]$ to $dir_{new}[7]$ at this point, having keys with prefix 0. To have bucket B2 at $dir_{new}[6]$, B1 splits again into two buckets, B1 and B3 in the figure 5(c). In the state of figure 5(c), the bucket B2 is covered by the bucket B3 and the bucket B1 does not need to split to make the bucket B2. Therefore, we fix the bucket B1 which is represented by green color. The bucket B3 splits to make B2, and the bucket B3 is fixed because the bucket B2 is not under the bucket B3. We can easily recover the split history by splitting bucket from beginning, but it requires to scan directory index for each index remapping. Since the remapped index is fixed from the new remapping index, we do this procedure by scanning the directory from beginning.

The bucket is pointed by $2^{(globaldepth-localdepth)}$ indexes of directory and we denote the number of indexes stride. Since we know that the stride is multiple of 2 and the remapped index, we try from smallest stride which is 1 until it reaches to the current remapped index or capacity of directory. For example, we first set the stride to 1 starting from $dir_{new}[0]$ at the state of figure 5(c) to build B2 and checks whether it can be doubled or not. The bucket B2 is connected to the $dir_{new}[6]$, so stride of $dir_{new}[0]$ is doubled twice covering four indexes. When it tries to double stride to 8, it covers $dir_{new}[6]$ which is not suppose to be covered. Therefore, stride for $dir_{new}[0]$ is fixed. The existance of bucket B1 with stride 4 requires its previous split which is represented as parent node in the figure 5(c) with the doubled stride and sibling with the same stride divided from the same parent node. We repeat it until the parent node is root and restart from index $dir_{new}[4]$ with the bucket B3 that is not fixed. The bucket B2 is under the bucket B3 and it cannot be pointed by the pointer that is not pointing the bucket B3. As we set the limitation of stride to the capacity, the limitation of stride of the bucket B2 is under the stride of bucket B3 of second level with local depth 2. The stride of the bucket B3 connected to the $dir_{new}[4]$ is 2 because $dir_{new}[6]$ is remapped index, so figure 5(d) shows the result to set the indexes with split history before the remapped index $dir_{new}[6]$.

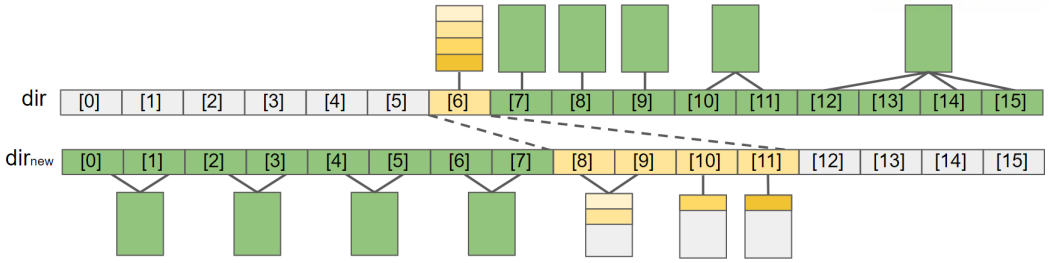


Figure 6: New directory adjuster

New directory adjuster

Figure 6 is one example state of the bucket adjuster. From the bucket pointed by dir[6], Directory adjuster remaps keys in the bucket with a new remapping function and insert the key into the appropriate bucket of the new directory. A tracking pointer is tracking the old directory index which is currently being adjusted for the foreground operations.

When a new key is inserted while remapping thread is adjusting the keys in the background, it converts the key to remapped key using remapping function and finds the index using a global depth prefix of the key. If the index is located to the left of the tracking pointer, it means that the remapping thread already finished the adjustment to the new directory. Therefore, foreground thread converts the key to remapped key again using the new remapping function and finds the index of the new directory.

VI Recovery

ScanHash ensures consistency of its hashing index upon recovery from failure. The recovery procedure of ScanHash is as follows. Similar to approaches in NoveLSM [12] and SLM-DB [13], ScanHash first recovers the root data structure that has pointers to the current directory, the new directory, and a structure to store the mean and variance of keys, etc., from the PM pool with the help of a PM manager [14]. Then, ScanHash invokes the directory recovery algorithm of CCEH for the current directory, and also for the new directory, if it exists. It then scans all the buckets in the old directory and in the new directory, if any, to collect any garbage key k by checking if the pointer of k is the same as that of its previous key. During this scan, ScanHash also counts the number of records for each bucket as ScanHash uses this count to efficiently perform hashing operations such as insertion. The recovery overhead for this full bucket scanning process is not excessive; for a full scan of 100 million keys in our experimental setup, it takes around 2 seconds.

If there is a new directory, this means that remapping was in progress. Thus, ScanHash needs to complete the remapping procedure. To restart remapping, ScanHash finds the largest key k stored in the new directory, which can be found during the above full scan of buckets. Then we search the old directory for this key. Given k is found in dir[w], we restart the adjuster to process from dir[w] based on the array of the local depth for dir_{new}, which was stored in the remapping log. Remapping is done in the background, and normal operations can resume ScanHash.

VII Evaluation

7.1 Methodology

For our experimental study, we use a machine with 32 Intel Xeon Hexa-core Gold 6242 processors (2.80GHz) with 20MB L3 Cache, 128GB DRAM and 496GB Intel Optane persistent memory. Ubuntu 18.10 with Linux kernel version 4.18 was installed in the machine.

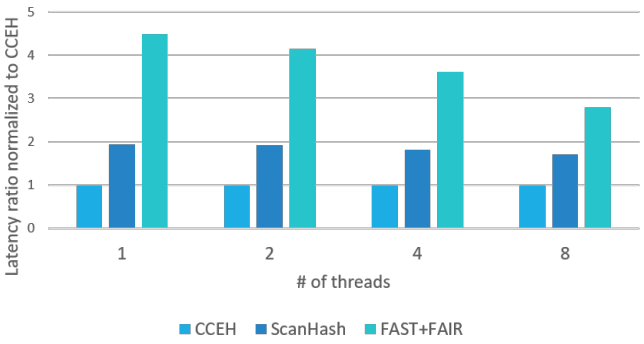
We use pmemobj library of PMDK to implement ScanHash. To evaluate the performance of ScanHash, we compare the performance of ScanHash to CCEH [4] with inplace update and FAST and FAIR B+-tree [3], denoted FFB. We modified CCEH and FFB on Github using the pmemobj library in the same way of ours. We use the default settings for CCEH which has 1024 slots for a single segment and FFB. ScanHash uses the same bucket size as a node of FFB which is 512 bytes. Since the CCEH does not provide deletion in their implementation on Github, we make a delete operation as an insert operation inserting invalid key without MFENCE because it does not update value pointer.

7.2 Microbenchmarks

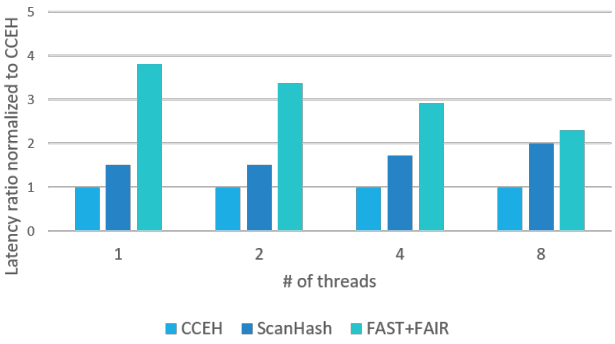
Figure 7 shows the point query latency ratio normalized to the CCEH. Deletion does not include the rebalance of the tree of FFB or directory updates of CCEH and ScanHash. In figure 7(a), insertion latency of ScanHash 1.94x slower than CCEH while FFB is 4.51x slower. This is because FFB has to travel tree from the root node to find the bucket for the key but ScanHash finds the bucket directly using the directory index. As the number of thread increases, the latency ratio of both FFB and ScanHash is getting close to each other because the internal node search can be done by multiple threads without a lock.

Since the rebalancing of the tree and table is not included in the deletion as shown in figure 7(b), the overhead to shift key and value pointer pairs is dominant overhead. While the performance of FFB is getting closer to the CCEH while multiple threads can concurrently search internal node, the performance improvement for the increasing number of threads in deletion is not effective for us because CCEH only get a lock for a cacheline size bucket in the segment but ScanHash has to lock the entire bucket to shift. The deletion latency of CCEH decreases 28.49% from 58.61 seconds for a single thread to 16.70 seconds for 8 threads while the deletion latency of ScanHash decreases 27.75% from 88.66 seconds for a single thread to 33.47 seconds for 8 threads.

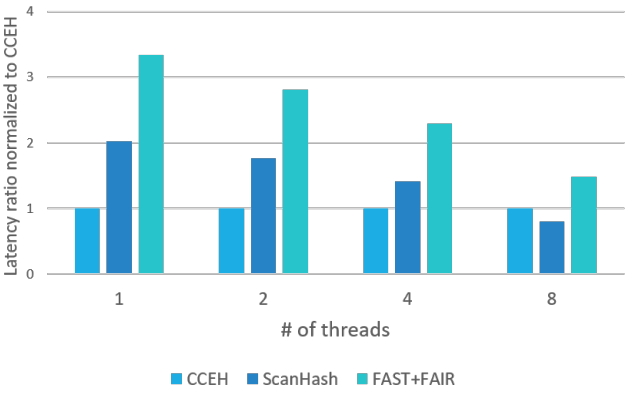
In figure 7(c), search latency of ScanHash and FFB is respectively 2.02x and 3.35x slower than CCEH for a single thread. Since search operation of CCEH with inplace update requires lock for the bucket split but FFB and ScanHash do not have a lock for search by checking the first key of sibling and key pattern respectively. As the number of thread increases, the latency of FFB and ScanHash is getting closer to the CCEH and the search performance of ScanHash for 8 thread is even better than CCEH because CCEH has a procedure to get and release lock



(a) Insertion latency ratio normalized to CCEH



(b) Deletion latency ratio normalized to CCEH



(c) Search latency ratio normalized to CCEH

Figure 7: Point query latency ratio of microbenchmarks with uniform distribution

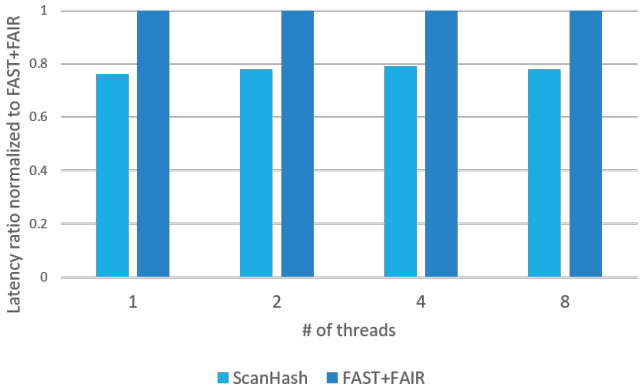


Figure 8: Range query latency ratio of microbenchmarks with uniform distribution

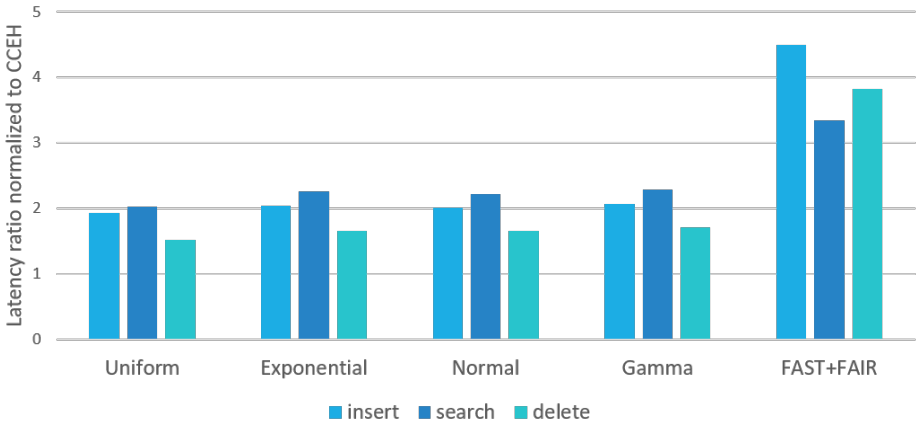


Figure 9: Latency ratio of microbenchmarks with non-uniform distribution

and ScanHash for the uniform distribution uses the identical function as a remapping function while hash function of CCEH does computation to uniformly distribute the key.

In figure 8, we compare the range query performance to the FFB only because CCEH does not provide the range query. We scan N keys from the given starting key or the smallest key bigger than the starting key if the key does not exist in the data structure. Since the range query of FFB is implemented to return the keys between first and last key on the Github, we modify the condition to finish the range query to make range query find N keys. As in the performance of search operation, FFB has to travel the tree to find the first node including the first key. After it reaches the leaf node, it scans leaf nodes with sibling pointer. Both ScanHash and FFB do not have a lock for search and scan, the performance of both increases, remaining the ratio normalized to FFB same.

7.3 Performance Over Non-Uniform Key Distributions

Key distributions are generated using the distribution generator of the random standard library. Figure 9 shows the latency ratio normalized to the CCEH of different distributions comparing to

the mean of FFB for a single thread. Overall performance decreases when the key distribution is not uniform because it computes to get remapped key using one of linear functions that approximate cumulative distribution function. Since the set of linear functions is divided by sub-range with the same number of keys, the first key of each sub-range is dynamically decided in the key distribution finder. Therefore, although the linear function itself is cheaper than the standard hash function, the overhead of remapping function is similar to the standard hash function because it scans the set of linear functions to find appropriate function. The search latency is 2.02x slower than CCEH for the uniform distribution and 2.21x, 2.20x, and 2.28x slower than CCEH for exponential, normal, and gamma distribution respectively.

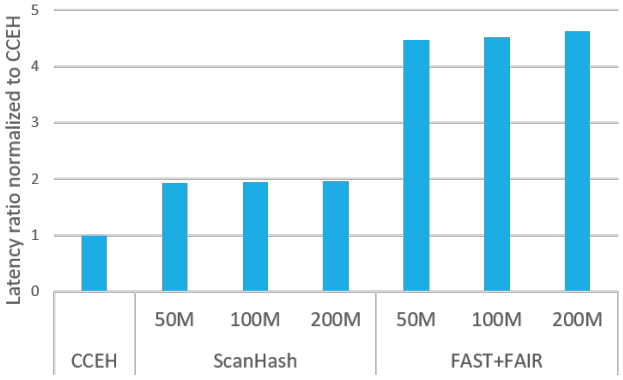
7.4 Scalability

Figure 10 and figure 11 show the latency ratio normalized to the CCEH when the number of keys is 50M, 100M, and 200M for the experimental result of the scalability. CCEH and ScanHash find an index of its directory to find the corresponding bucket, but FFB has to scan the internal nodes to find the corresponding leaf node. In the CCEH and ScanHash, the size of the directory and the number of buckets affects the performance in terms of the cache hit. In FFB, an additional depth of tree makes a single query to scan one more internal node. In figure 10(a), the latency ratio normalized to CCEH per key decreases 9.87% on average when the number of the key is doubled because of the tree rebalancing and shift overhead which is dominant hides the tree traversal overhead. In figure 10(b), since the delete operation does not include tree rebalancing overhead, the latency ratio normalized to CCEH per key decreases 16.03% on average when the number of the key is doubled. In figure 10(c), 27.45% on average when the number of the key is doubled while the latency ratio normalized to CCEH per key remains because it shows a similar tendency to the CCEH. For the range query, the latency ratio normalized to the FFB is getting smaller as the number of key increases because of the search latency to find the bucket with starting key as shown in figure 11.

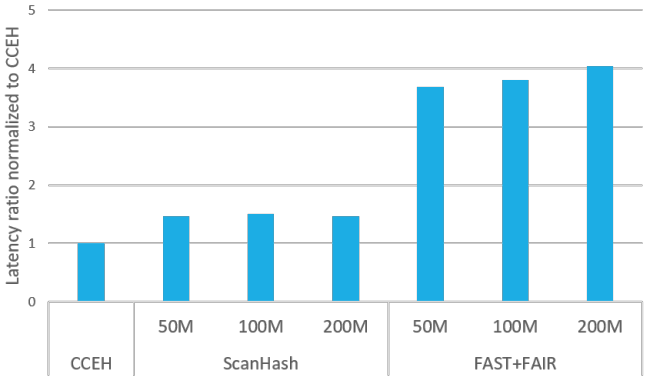
7.5 Size of Bucket

If the bucket has more slot, the overhead to shift and scan increases but the overhead to split the bucket decreases. We do a sensitivity experiment for the size of a bucket that is directly related to the performance fo ScanHash. Figure 12 shows the experimental result of uniform distribution using 256 bytes, 512 bytes, and 1024 bytes of bucket size.

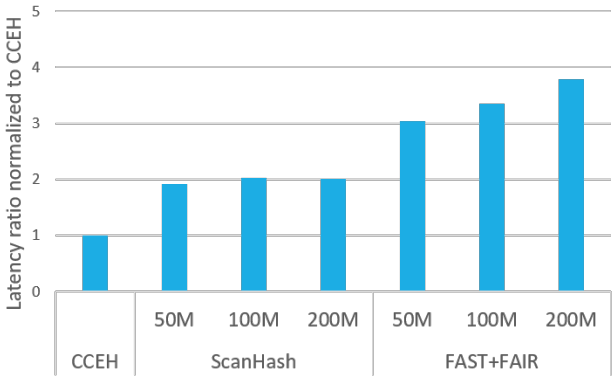
Each bar represents the latency ratio normalized to the default bucket size which is 512 bytes having 30 slots for key and value pairs because of its metadata. Each key and value pointer pair is 16 bytes, so 4 key and value pointer pair is in a single cacheline in our experiment. The bucket with 512 bytes is consisting of 8 cachelines and the bucket with 256 bytes and 1024 bytes are 4 and 16 cachelines respectively. Latency to insert the key in the 1024 bytes bucket which has 62 slots increases 18.11% comparing to the default but latency to insert in the 256 bytes



(a) Insert latency ratio normalized to CCEH



(b) Delete latency ratio normalized to CCEH



(c) Search latency ratio normalized to CCEH

Figure 10: Point query latency of microbenchmarks varying number of keys

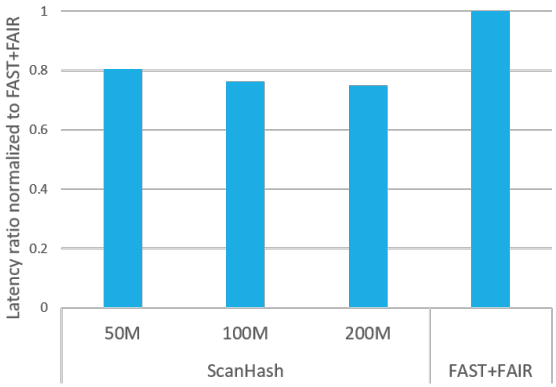
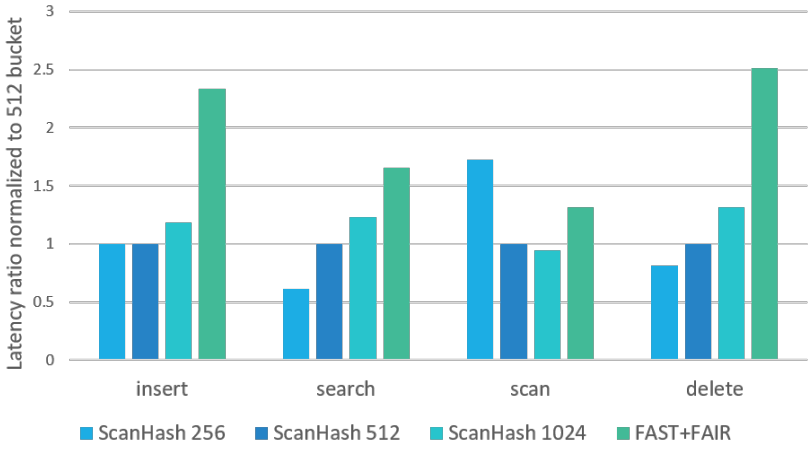


Figure 11: Range query latency of microbenchmarks varying number of keys



(a) Latency ratio normalized to default bucket size (512)

Figure 12: Latency of microbenchmarks varying size of bucket

bucket which has 14 slots does not decrease because the 256 bytes bucket trigger bucket split and directory doubling frequently comparing to the others. Search latency of ScanHash with 256 bytes bucket decreases to 61.34% than the 512 bytes bucket while search latency of 1024 bytes bucket increases 22.83%. Since the 256 bytes bucket can only contain 14 keys, it necessarily has a larger directory and more buckets to store the same amount of key. Therefore, scan operation reads more bucket traveling more sibling pointers that make the performance to scan slower than the FFB, 72.48% slower than the ScanHash default bucket size while FFB is 31.30% slower. On the other hand, the 1024 bytes bucket is faster to scan than the 512 bytes bucket.

7.6 Yahoo! Cloud Serving Benchmarks(YCSB)

Yahoo! Cloud Serving Benchmarks, denoted YSCB, is one of the benchmarks that is considered as a real-world workload. In our experiments, we use the default configuration of YCSB workloads with 1 million operations and execute workloads suggested order in the manual on Github. The workloads are continuously executed in order of workload A of 50% reads and 50% updates after

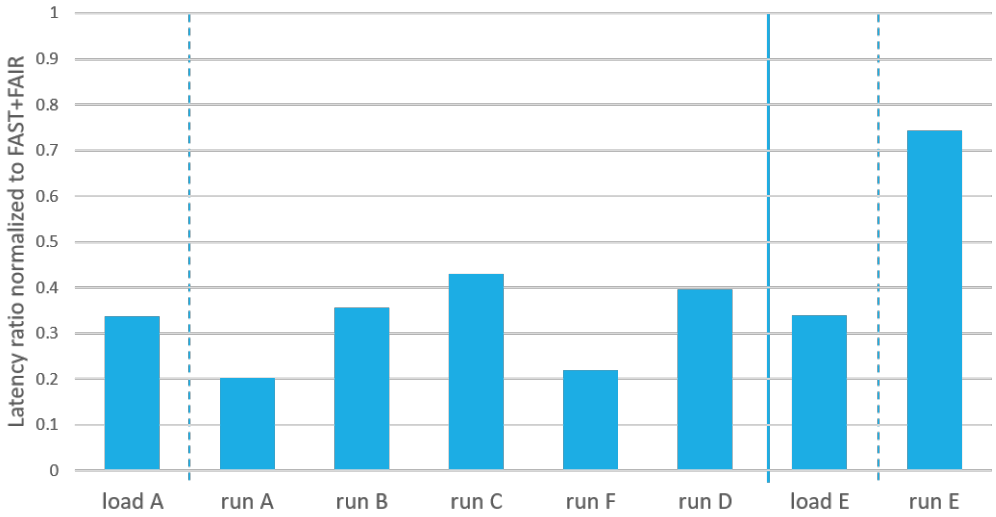


Figure 13: Latency ratio of YCSB workloads with uniform distribution

loading the database, workload B of 95% reads and 5% updates, workload C of 100% reads, workload F of 50% reads and 50% read-modify-writes, where Zipfian distribution is used. Then, workload D of 95% reads for the latest keys and 5% inserts are executed. Workload From A to F except for E is for the point query operation. After loading the database again, workload E of 95% range query and 5% inserts with Zipfian distribution is executed. For workload E, the scan size is set to 100. The overall performance is better than the FFB. During the execution, the key distribution checker does not trigger remapping and when we analyze the key pattern generated by YCSB, the key itself is generated randomly which means that the key distribution is uniform. The zipfian key pattern represents frequency to access a certain key. It includes the update operation which is different from the insert operation, so we set the update operation to searches the key first and insert the key into the bucket if the key exists. Since the in-place update of CCEH implementation on GitHub has a problem, we compare the performance to the FFB.

VIII Conclusion

In this work, we proposed the scannable hash table, ScanHash, that provides a fast point query using the benefits of the hash table while supporting the range query. ScanHash uses a remapped key instead of a hash key to distribute key uniformly and preserves the sorted order of key in the bucket to provide range query efficiently using the FAST algorithm. Although key remapping is under the assumption of known distributions, any distribution can be added with the cumulative distribution function and information of its parameter and characteristic. In our experimental result, we demonstrate the performance of ScanHash is better than B+-tree in both point query and range query.

References

[1] F. Xia, D. Jiang, J. Xiong, and N. Sun, "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC)*, 2017.

[2] W. Ma, Y. Zhu, C. Li, M. Guo, and Y. Bao, "Bilokey : A scalable bi-index locality-aware in-memory key-value store," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 7, pp. 1528–1540, Jul. 2019.

[3] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree," in *Proceedings of the 16th Usenix Conference on File and Storage Technologies (FAST)*, 2018.

[4] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-Optimized Dynamic Hashing for Persistent Memory," in *Proceedings of the 17th Usenix Conference on File and Storage Technologies (FAST)*, 2019.

[5] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing—a fast access method for dynamic files," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, Sep. 1979. [Online]. Available: http://doi.acm.org/10.1145/320083.320092

[6] H. . P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec 2010.

[7] Y. Huai, "Spin-transfer torque MRAM (STT-MRAM): Challenges and prospects," *AAPPS bulletin*, vol. 18, no. 6, pp. 33–40, Dec. 2008.

[8] Intel, "Optane DC persistent memory," https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[9] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh, "WORT: Write optimal radix tree for persistent memory storage systems," in *15th USENIX Conference on File and Storage Technologies (FAST)*, 2017. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/lee-se-kwon

[10] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems," in *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[11] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, Feb. 2015. [Online]. Available: http://dx.doi.org/10.14778/2752939.2752947

[12] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/kannan

[13] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. Choi, "SLM-DB: Single-Level Key-Value Store with Persistent Memory," in *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[14] "pmem.io Persistent Memory Programming," https://pmem.io/.

Acknowledgements

Thank you for recognizing my graduation achievement.